**Significant Transitions**

- **Programming languages (PLs)**
  - They evolve slowly and occasionally, e.g.:
    - *C to C++*: More robust data structures (objects)
    - *C++ to Java*: More robust control flows (strong typing)
  - But new *programming models* are invented routinely
    - As domain-specific libraries or API’s
    - As program analysis tools
    - As language extensions

- **Transitions**
  - Significant transitions in programming models eventually “precipitate” into new programming languages (unpredictably)
  - We can watch out for significant transitions in programming models
Transitions in 3 (related) areas

• We are in the middle of a radical transition in programming models (and eventually PLs)
• A new emphasis on computation on WANs
  - Wide area flows
    • Messages nor RPC, schedules not threads. *Messaging API’s.*
    • Need to integrate these new flows into PL control constructs.
  - Wide area data
    • XML is “net data”. *XML API’s.*
    • Need to integrate this new data into PL data structures.
  - Wide area protection
    • Access control, data protection. *Security and privacy API’s.*
    • Need to integrate security properties into PL assertions.

• Disruptive transitions
  - Forget RPC (and threads): the world is asynchronous.
  - Forget type systems as we know them.
  - Forget trusting anything non-local.
Flow Integration

• Wouldn’t it be nice to hide concurrency from programmers?
  - SQL does it well
  - UI packages do it fine (mostly single-threaded!)
  - RPC does it ok
  - But we are moving towards more asynchrony, i.e. towards more visible concurrency (e-commerce scripts and languages, web services, etc.)

  You can hide all concurrency some of the time, and you can hide some concurrency all the time, but you can’t hide all concurrency all the time.

  - Asynchronous message-based concurrency does not fit easily with more traditional shared-memory synchronous concurrency control

  • Goal: make concurrent flows available and checkable at the language level.
Data Integration

• Wouldn't it be nice to "program directly against the schema" in a well-typed way?
  - PL data has traditionally been "triangular" (trees), while persistent data has traditionally been "square" (tables)
  - This has caused integration problems: the “impedance mismatch” in data base programming languages
  - Now, persistent data (XML) is triangular too!
  - However, the type systems for PL data (based on tree matching) and XML (based on tree automata) are still deeply incompatible

• Goal: make semistructured data easily available and checkable at the language level.
Protection Integration

• Wouldn’t it be nice to have automatic security?
  - It’s an applet. Sits in a sandbox. End of story. (?)
  - Ok, what about semi-automatic security? Explicitly grant/require permissions. (Stack walking etc.)
  - Leads to “sophisticated” access models that programmers do not understand reliably.

• Security today: obscure mechanisms to prevent something from happening.
  - It is usually not clear what security mechanisms are meant to achieve.
  - Need to move towards declarative security and privacy interfaces and policies.

• Goal: make protection policies available and checkable at the language level.
Language Reliability

• Whether or not we merge new programming models into PLs, we need analysis tools for these new situations
  - Flow: e.g.: behavioral type/analysis system
    • “Does the program respect the protocol?”
  - Data: e.g.: semistructured type/analysis systems
    • “Does the program output match the schema?”
  - Protection: e.g.: information-flow type/analysis system
    • “Does the program defy policy or leak secrets”

• Analysis tools are critical for software reliability
  - Getting it right without assistance is just too hard.
  - These technologies need to be developed in any case.
A Personal Agenda

• **Flows** [exploit join calculus]
  - Synchronization chords
  - $\mathcal{C}_\omega$ (f.k.a. Polyphonic C#)

• **Data** [exploit spatial logics as types]
  - Description logics
  - $\mathcal{C}_\omega$ (f.k.a. Xen/X#)

• **Protection** [exploit $\pi$-calculus-style restriction]
  - Flows: Secrecy and Group Creation
  - Data: Trees with hidden labels
Flows
Language Support for (WAN) Distribution

• Distribution ⇒ concurrency + latency
  ⇒ asynchrony
  ⇒ more concurrency
  - Approaches: Message-passing, event-based programming, dataflow models, etc.
  - Languages: coordination (orchestration) languages, workflow languages, etc.

• Good language support for asynchrony
  - Make invariants and intentions more apparent (part of the interface), because:
    • It’s good software engineering
    • Allows the compiler much more freedom to choose different implementations
    • Also helps other tools
• An extension of the C# language with new concurrency constructs
• Based on the join calculus
  - A foundational process calculus like the \( \pi \)-calculus but better suited to asynchronous, distributed systems.
  - First applied to functional languages (JoCaml).
  - It adapts remarkably well to o-o classes and methods.
• A single model that works for
  - Local concurrency (multiple threads on a single machine).
  - Distributed concurrency (asynchronous messaging over LAN or WAN).
  - With no distributed consensus.
• It an unusual model. But it’s also a simple extension of familiar o-o notions.
  - No threads, no locks, no fork, only join.
In one slide:

- **Client Side (method invocation)**
  - Objects have both *synchronous* and *asynchronous* methods.
  - If the method is synchronous, the caller blocks until the method returns some result (as usual).
  - If the method is *async*, the call completes at once and returns void (as in message passing).

- **Server Side (class definition)**
  - A class defines a collection of *chords* (method synchronization patterns), which define what happens once a particular *set* of methods have been invoked. One method may appear in several chords.
  - When enough pending method calls match a chord pattern, the chord body runs. If there are several matches, an unspecified chord is selected.
  - Each chord can have *at most* one synchronous method (providing the *result*). A chord containing *only* asynchronous methods effectively forks a new thread.
A simple unbounded buffer

class Buffer {
    String get() & async put(String s) {
        return s;
    }
}
A simple unbounded buffer

```java
class Buffer {
    String get() & async put(String s) {
        return s;
    }
}
```

• An ordinary (synchronous) method header with no arguments, returning a string
A simple unbounded buffer

class Buffer {
    String get() & async put(String s) {
        return s;
    }
}

• An ordinary (synchronous) method header with no arguments, returning a string

• An asynchronous method header (hence returning no result), with a string argument
A simple unbounded buffer

class Buffer {
    String get() & async put(String s) {
        return s;
    }
}

• An ordinary (synchronous) method header with no arguments, returning a string
• An asynchronous method header (hence returning no result), with a string argument
• Joined together in a chord with a single body
• Calls to `put()` return immediately (but are internally queued if there’s no waiting `get()`).

• Calls to `get()` block until/unless there’s a matching `put()`

• When there’s a match the body runs, returning the argument of the `put()` to the caller of `get()`.

• Exactly which pairs of calls are matched up is unspecified.

```java
class Buffer {
    String get() & async put(String s) {
        return s;
    }
}
```
A simple unbounded buffer

```java
class Buffer {
    String get() & async put(String s) {
        return s;
    }
}
```

- Does this example involve spawning any threads?
  - No. Though the calls will usually come from different pre-existing threads.
- So is it thread-safe? You don’t seem to have locked anything…
  - Yes. The chord compiles into code which uses locks. (And that doesn’t mean everything is synchronized on the object.)
- Which method gets the returned result?
  - The synchronous one. And there can be at most one of those in a chord.
VAR i: INTEGER;
VAR m: Thread.Mutex;
VAR c: Thread.Condition;

PROCEDURE AcquireExclusive();
BEGIN
   LOCK m DO
       WHILE i # 0 DO Thread.Wait(m,c) END;
       i := -1;
   END;
END AcquireExclusive;

PROCEDURE AcquireShared();
BEGIN
   LOCK m DO
       WHILE i < 0 DO Thread.Wait(m,c) END;
       i := i+1;
   END;
END AcquireShared;

PROCEDURE ReleaseExclusive();
BEGIN
   LOCK m DO
       i := 0; Thread.Broadcast(c);
   END;
END ReleaseExclusive;

PROCEDURE ReleaseShared();
BEGIN
   LOCK m DO
       i := i-1;
       IF i = 0 THEN Thread.Signal(c) END;
   END;
END ReleaseShared;

An integer i represents the lock state:

-1 ↔ 0 ↔ 1 ↔ 2 ↔ 3 ...
(exclusive) (available) (shared)
public class ReaderWriter {
    public void AcquireExclusive() & async Idle() {}
    public void ReleaseExclusive() { Idle(); }

    public void AcquireShared() & async Idle() { S(1); }
    public void AcquireShared() & async S(int n) { S(n+1); }
    public void ReleaseShared() & async S(int n) {
        if (n == 1) Idle(); else S(n-1);
    }

    public ReaderWriter() { Idle(); }
}

A single private message represents the state:

\[
\begin{array}{c|c|c|c}
none & \leftrightarrow & Idle() & \leftrightarrow S(1) & \leftrightarrow S(2) & \leftrightarrow S(3) & \ldots \\
(exclusive) & (available) & (shared)
\end{array}
\]

A pretty transparent description of a simple state machine. Moreover, the synchronization patterns are apparent in the class interface,
Features

- A clean, simple, new model for asynchronous concurrency
  - Minimalist design – to build whatever complex synchronization behaviors you need
  - Easier to express and enforce concurrency invariants; not “buried in the code” any more
  - Much better than programming reactive state machines by hand (the compiler does it for you).
  - Efficiently compiled to queues, automata, match bit-vectors, and thread pools.
  - Compatible with existing constructs, though they constrain our design somewhat
  - Solid foundations, on which to build analysis tools.
Ongoing Work

• Protocol contracts
  - Typechecking-style support for checking the interaction of concurrent protocols.
  - A.k.a behavioral type system, session types, etc.

• Required for software reliability

• Facilitated by explicit concurrency interfaces.
Data
DATA

Semistructured Data
(I.e.: XML after parsing)

Articles

Paper

Author Title Year

A B C

G D K

• A tree (or graph), unordered (or ordered). With labels on the edges.
• Invented for “flexible” data representation, for quasi-regular data like address books and bibliographies.
• Adopted by the DB community as a solution to the “database merge” problem: merging databases from uncoordinated (web) sources.
• Adopted by W3C as “web data”, then by everybody else.
It's Unusual Data

• Not really arrays/lists:
  - Many children with the same label, instead of indexed children.
  - Mixture of repeated and non repeated labels under a node.

• Not really records:
  - Many children with the same label.
  - Missing/additional fields with no tagging information.

• Not really variants (tagged unions):
  - Labeled but untagged unions.

• Unusual data.
  - Yet, it aims to be the new universal standard for interoperability of programming languages, databases, e-commerce...
Needs Unusual Languages

• New *flexible* types and schemas are required.
  - Based on “regular expressions over trees” reviving techniques from tree-automata theory.

• New processing languages required.
  - Xduce [Pierce, Hosoya], Cduce, ...
  - Various web scripting abominations.

• New access methods/query languages required.
  - E.g. Existence of paths through the tree.
Data Descriptions

• We want to *talk about* data
  - I.e., specify/query/constrain/typecheck the possible structure of data, for many possible reasons:
    • Typing (and typechecking): for language and database use.
    • Constraining (and checking): for policy or integrity use.
    • Querying (and searching): for semistructured database use.
    • Specifying (and verifying): for architecture or design documents.

• A *description* (*spatial formula*) is a formal way of talking about the possible structure of data.
  - We go after a general framework: a very expressive language of descriptions.
  - Combining logical and structural connectives.
  - Special classes of descriptions can be used as types, schemas, constraints, queries, and specifications.
In Cambridge there is (at least) a pub called the Eagle that contains (at least) one empty chair.

In Cambridge there is (nothing but) a pub called the Eagle that contains (nothing but) two empty chairs.

Data

Cambridge[
  Eagle[
    chair[0] | chair[0]
  ]
]

Description

Cambridge[
  Eagle[
    chair[0] | T
  ] | T
]

data matches description
Example: Queries

With match variables $\mathcal{X}$: Who is really sitting at the Eagle?

$$\begin{align*}
\text{Eagle[} & \text{Yes: } \mathcal{X} = John[0] \\
\text{chair[} & \text{Yes: } \mathcal{X} = Mary[0] \\
\text{T] & \\
\text{]} &
\end{align*}$$

With select-from:

$$\begin{align*}
\text{from Eagle[...] & Single result:} \\
\text{match Eagle[chair[} & \text{person[John[0]] |} \\
\text{chair[} & \text{person[Mary[0]]} \\
\text{0 \land \mathcal{X}] | T]} & \\
\text{select person[\mathcal{X}]} &
\end{align*}$$
“Vertical” implications about nesting

\[ \text{Borders}[^\text{T}] \Rightarrow \text{Borders}[\text{Starbucks}[^\text{T}] | \text{T}] \]

If it’s a Borders, then it must contain a Starbucks

“Business Policy”

“Horizontal” implications about proximity

\[ (\text{NonSmoker}[^\text{T}] | \text{T}) \Rightarrow (\text{Smoker}[^\text{T}] | \text{T}) \]

If there is a NonSmoker, then there must be a Smoker nearby

“Social Policy”
**Example: Schemas**

- Descriptions are a “very rich type system”. We can comfortably represent various kinds of schemas.
- Ex.: Xduce-like (DTD-like) schemas:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>the empty tree</td>
</tr>
<tr>
<td>$\mathcal{A} \mid \mathcal{B}$</td>
<td>an $\mathcal{A}$ next to a $\mathcal{B}$</td>
</tr>
<tr>
<td>$\mathcal{A} \lor \mathcal{B}$</td>
<td>either an $\mathcal{A}$ or a $\mathcal{B}$</td>
</tr>
<tr>
<td>$n[\mathcal{A}]$</td>
<td>an edge $n$ leading to an $\mathcal{A}$</td>
</tr>
<tr>
<td>$\mathcal{A}^*$</td>
<td>$\mu X.0 \lor (\mathcal{A} \mid X)$ the merge of zero or more $\mathcal{A}$s</td>
</tr>
<tr>
<td>$\mathcal{A}^+$</td>
<td>$\mathcal{A} \mid \mathcal{A}^*$ the merge of one or more $\mathcal{A}$s</td>
</tr>
<tr>
<td>$\mathcal{A}?$</td>
<td>$0 \lor \mathcal{A}$ zero or one $\mathcal{A}$</td>
</tr>
</tbody>
</table>
Ongoing Work

• Freely mixing logic and data: spatial logics
  \[ a[b[A \lor B]] \Rightarrow a[C] \]
  - Can be seen as type systems, query languages, policy specifications, etc.
  - Special cases are regular expressions over trees (XML query, etc.)
  - Lots of open theoretical problems in this area (typing and subtyping algorithms, decidable sublogics, etc.)
\[ T ::= N \mid T[\ ] \mid T\{\} \mid T(..., T, ...) \mid T \mid T \mid T^* \mid [..., T_{m}, ...] \]
Protection
Hiding

• Any kind of security/privacy issue has to do with hiding something
  - Hiding procedures by access control
  - Hiding data by encryption
• In programming languages:
  - How can we protect/hide flows? (Security)
  - How can we protect/hide data? (Privacy)
  - Exploit the mother of all hiding operators: \( \pi \)-calculus restriction (already widely used in crypto protocol analysis).
Flow Protection: Group Creation

- *Group creation* is a new general construct that can be added to virtually any language or formalism.
- It is a natural extension of the sort-based type systems developed for the \(\pi\)-calculus:
  - \((\forall G) (\forall n:G)(\forall m:G) \ldots\)
  - create a new group (i.e., unstructured type or collection) \(G\), and populate it with new elements \(n, m, \ldots\)
- A secret like \(n\) can never escape from the initial scope of \(G\), as a simple matter of typechecking.
Untrusted Opponents

- Problem: opponents cheat. Suppose the opponent is untyped, or not well-typed (e.g.: running on an untrusted machine):

\[
(\forall p: U) \quad (p(y).O' \mid (\forall G) (\forall x: G) (p(x) \mid P'))
\]

- Will an untyped opponent, by cheating on the type of the public channel \( p \), be able to acquire secret information?

- Fortunately, no. The fact that the player is well-typed is sufficient to ensure secrecy, even in presence of untyped opponents. Essentially because \( p(x) \) must be locally well-typed.

- We do not even need to trust the type of the public channel \( p \), obtained from a potentially untrusted name server.
Secrecy Guarantee

• **Programmer’s reference manual:**
  Names of group $G$ remain *secret*, forever, outside the initial scope of $(\forall G)$.

• **Secrecy Theorem (paraphrased)**
  If $(\forall G)(\forall x:\ldots G\ldots)P$ is well-typed, then $P$ will not leak $x$ even to an untyped (untrusted) opponent.
\[ P, Q ::= \]
\[ 0 \]
\[ n[P] \]
\[ P \mid Q \]
\[ (\forall n)P \]
Tree Equivalence (Structural Congruence)

• $(\forall n)(P \mid (\forall n)Q) \equiv ((\forall n)P) \mid ((\forall n)Q)$

• $(\forall n)m[P] \equiv m[(\forall n)P]$ if $n \neq m$
Ex: Local Pointers

- E.g., XML IDREFs

Encoded as (unique) $addr[y[0]]$

Anonymous pointer id

Encoded as $ptr[y[0]]$

$y$

 addr

 ptr
  y

$y$
Ex: Unique and Unguessable IDs

an account

anonymous account number

checkbook

another account

another (guaranteed different) account number
Type Systems for Hidden Names

- **account**: `H y. ... id[y] ... checkbook[y] ...`
  - Hiding quantifier

- These are *name-dependent* types
  - Dependent types: traditionally very hard to handle because of computational effects.
  - But dependent only on “pure names”: no computational effects.
  - Name-dependent types are emerging as a general techniques for handling freshness, hiding, protocols (e.g. Vault), and perhaps security/privacy aspects in type systems.
Conclusions
WAN Flows, Data, Protection

• New languages
  - Language evolution is driven by wishes.
  - Language adoption is driven by needs.

• We now badly need evolution in areas related to WAN-programming for non-experts (i.e. with language support).
  - Concurrent flows.
    • Applications of Join Calculus.
  - Semistructured data.
    • Applications of Spatial Logics.
  - Flow and data protection.
    • Applications of $\pi$-calculus restriction.
References

- **Flows**
  - *Join Calculus*: Fournet *et al.*
  - *Polyphonic C#*: Benton, Cardelli, Fournet.
  - *Behave!*: Larus *et al.* *Vault*: DeLine *et al.*
  - + Kobayashi, Honda, Yoshida, Vasconcelos, ...

- **Data**
  - *TQL*: Cardelli, Ghelli *et al.*

- **Protection**
  - *Fresh-ML*: Pitts *et al.*
  - *Secrecy and Groups*: Cardelli, Ghelli, Gordon.
  - *Trees with Hidden Labels*: Cardelli, Gardner, Ghelli.

*(See personal web pages or search engines.)*